Edge Hill University

The Department of Computer Science

The Development of a Video Game using Procedural Terrain Generation

CIS3140 CW2 – Project Report

Student: James Edwards

Supervisor: Saad Saihi

Date: 6 May 2025

This Report is submitted in partial fulfilment of the requirements for the BSc Honours Computing (Games Programming) Degree at Edge Hill University.

Abstract

This project explores the creation of a small video game that is centred around the use of procedural terrain generation. The achievement in this project is creating procedural generated terrain using two noise algorithms, Fractal Perlin Noise and Worley Noise. The Fractal Perlin noise algorithm was used in this project to create the base structure of the mesh as well as rocky formations in the terrain and the Worley Noise terrain was used to generate ridges in the terrain to enhance the detail of the terrain. These noise functions are showcased in 4 different levels in the game. Each level showcases a different noise technique. One level uses standard perlin noise, one uses fractal perlin noise, one uses Worley noise, and the final level uses a combination of fractal and perlin noise. Additionally, an attempt was made to implement spline-based terrain generation into the project.

Additionally, a physics-based skier character controller can be used to play the levels in the game, this character features a fully rigged and animated character model.

Finally, a summary of cutting-edge procedural terrain generation techniques is showcased in the literature review section of this report.

Acknowledgments

Firstly, I would like to thank my dissertation supervisor Saad Saihi for advising me and providing help complete this dissertation project.

I would also like to thank Kyle Worrall for his mentorship throughout the 3rd year of my undergraduate degree.

Thank you to my Mum, Dad and Sisters for always being there for me and dealing with me playing video games until late into the night. I can't believe that the passion I had for playing video games has now turned into a degree.

Lastly, I would like to thank my girlfriend Niamh for her support and encouragement throughout my entire degree. I couldn't have done it with you.

Table of Contents

Table of Figures

Chapter 1: Introduction

Procedural content generation is a field in game development where content such as 3D models, levels, etc is generated via the use of algorithms (Liu et al., 2019; Summerville et al., 2018). Many games such as Minecraft (Mojang, 2011), No Mans Sky (Hello Games, 2016) and Bad North (Stålberg et al., 2018) use Procedural content generation. Procedural content generation is useful to game studios as it helps reduce the man hours involved from artists and designers (Summerville et al., 2018) which allows the studio to reduce costs as well as increase the amount of content in a game. This will increase the replay ability of the game(Summerville et al., 2018).

This project primarily focuses on the procedural content generation of gameplay focused terrain. This focus has been chosen as the majority of research on procedural terrain generation algorithm focuses on generating realistic and aesthetically pleasing terrain but does not go into detail on how the terrain affects the gameplay elements in the game (Pech, Lam and Masek, 2020). To do this, this project will be programmed in C++ using Unreal Engine 5 and use make use of Unreal Landscape spline component to create paths on a mountain that can be used for a skiing video game.

1.1 Motivations for Project

When analysing literature, a common issue with the current research in the field of procedural content generation was found. This issue is that many forms of procedural content generation only focus on generating terrain that looks realistic and aesthetically pleasing (Huang and Yuan, 2023; Su et al., 2024; Golubev, Zagarskikh and Karsakov, 2016) but does not focus on how gameplay elements would be implemented with the new procedural content generation algorithm that papers are discovering and inventing (Pech, Lam and Masek, 2020). This work attempts to fill this gap by creating a video game that uses procedural terrain generation to generate the levels in the game. As there has already been a large amount of research on creating mountainous terrain (Huang and Yuan, 2023; Su et al., 2024; Golubev, Zagarskikh and Karsakov, 2016) this game will be set on mountainous terrain and use some of the algorithms that is already being commonly used for procedural terrain generation such as the perlin noise algorithm. Due to the setting of the game being on a mountain, this game will be a skiing game. Additionally, because many of the algorithms generate realistic terrain the skiing character should be Physics-based to match this realism.

1.2 Aim and Objectives

The Aim of the project is to: Create a skiing game that uses procedural content generation (PCG) to generate the terrain and the levels.

No	Objective Title	Objective Description
1	Create procedural skiing mountain terrain using PCG techniques.	Create a mountain by first procedurally generating a heightmap using PCG techniques such as fractal perlin noise then applying this heightmap to an Unreal Engine Landscape.
2	Sculpt terrain using Unreal Engine Terrain Splines.	Create a procedurally generated spline path and then apply this path to the landscape generated in objective 1. This spline path can later be used in objective 4 and 5.
3	Create a physics-based skiing character controller.	Create a physics-based skiing character controller that will ski down the procedural landscape.
4	Implement gameplay mechanics such as checkpoints, level start and finish and high score.	Add a start and finish line to the game as well as a checkpoints system to make a complete game. Also add a high score system that allows players to compete for the best time.
5	Add polish to the game by adding a user interface	Add polish to the game by creating a main menu and a heads-up display

FIGURE 1 PROJECT AIMS AND OBJECTIVES

Chapter 2: Background / Literature Review

2.1 A Review of Noise based PCG Techniques

One common technique for procedurally generating terrain by generating 2D noise and using it as a heightmap for a terrain mesh (Hyttinen, Mäkinen and Poranen, 2017). There are various noise algorithms that can be used to generate this 2D noise such as the value, perlin and Worley noise algorithms.

2.1.1 Perlin Noise

The perlin noise algorithm is an algorithm invented by Ken Perlin in his seminal paper 'An Image Synthesizer' (Perlin, 1985). This algorithm generates pseudo-random numbers between two values that gradually increase and decrease. These values are often values between 0.0 and 1.0 (Unity, N.d; Hart, 2001) but some perlin noise algorithms such as the algorithm used by Unreal Engine generate values between -1.0 and 1.0 (Epic Games, N.d).

This algorithm starts by taking both an X and a Y floating-point number as an input and then finds the nearest 4 corners that they fit into on a 2d grid of integers (Raouf, N.d). For Example, if the input was (3.3, 5.8) the corners would be (3,5), (4,5), (3,6) and (4,6). Then a pseudo random direction vector will be assigned to each one of these points This vector will often be generated using some form of hashing function (Kora, 2007) (Pseudo random directions are used so that the direction will be the same each time the perlin noise function is called (Dogo's Science 2, 2024)). Once the direction vectors have been set the dot product between the gradient vector at each corner and the offset vector (Offset Vector = (X, Y) - (CornerX, CornerY)) will be calculated. These dot products are used to quantify the influence of each corner's direction vector on the point. Finally, the dot products are interpolated together in order to produce a smooth noise value. Then they are interpolated linearly (Dogo's Science 2, 2024). Perlin noise is control via two parameters, amplitude that controls the strength of the perlin noise (Etherington, 2022) as well as frequency that controls how many peaks and valleys their will be in the noise (Etherington, 2022).

This perlin noise function can be used to create a 2D image heightmap as shown in the image below.

FIGURE 2 PERLIN NOISE 2D IMAGE (UNITY, N.D)

The perlin noise in this image has been generated using Unity's in built perlin noise function (Unity, N.d) although both Unity and Unreal Engine have an in-built Perlin noise function (Unity, N.d; Epic Games, N.d). This image can be used as a height map on a 3d terrain mesh. The image below is an example of a 2D height map with the perlin noise function applied to it.

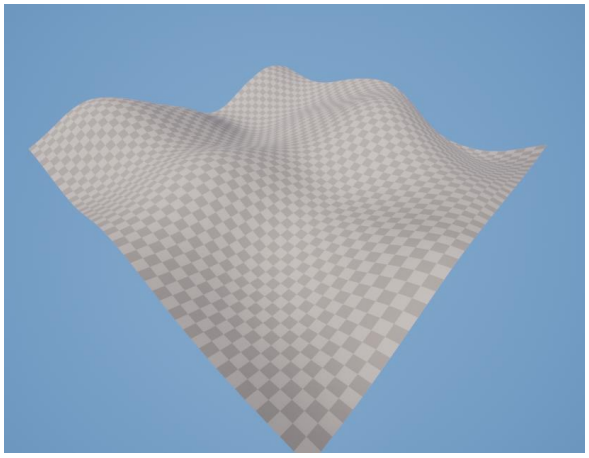

FIGURE 3 PERLIN NOISE BASED TERRAIN MESH

Many popular video games have used Perlin Noise in order to generate their terrain procedurally. One example of perlin noise being used is the survival and crafting game, Minecraft (Mojang, 2011). Two major benefits of the fractal noise algorithm is that it is infinitely tile able and it can be recreated by just using the same Frequency and Amplitude variables. Minecraft (Mojang, 2011) uses this so that the entire terrain of its endless map doesn't have to be saved when the player creates a new world. Additionally, Minecraft (Mojang, 2011) uses a chunk-based system that doesn't require the entire terrain to be loaded at once just the chunks that the player is in and nearby.

2.1.2 Fractal Noise

While perlin noise can create smooth looking terrain it doesn't generate terrain that looks as rough as real-world terrain. One technique that is used in order to resolve this issue is Fractal Noise. Fractal Noise is when multiple layers of noise are added on top of each other in order to generate a more complex and detailed noise heightmap. These multiple layers of noise are known as octaves (Lague, 2016). Typically, each octave of noise has a lower amplitude and higher frequency then the previous layer of noise. In most implementations of fractal noise, persistence and lacunarity variables are used to control how much lower the amplitude of the noise and how much higher the frequency of the noise is for each octave. Persistence variable controls how much the drop off in the amplitude is (how persistent it is) and Lacunarity is used to control how much the increase in frequency is for each octave of noise (Lague, 2016).

Below is an example of a terrain heightmap that uses Fractal Perlin Noise.

FIGURE 4 FRACTAL PERLIN NOISE HEIGHTMAP (DUNN, N.D)

2.1.3 Worley Noise

Another algorithm for generating noise is the Worley Noise algorithm (also known as the Cellular Noise algorithm). The Worley Noise algorithm is a noise function created by Steven Worley in his 1996 player "A Cellular Texture Basis Function" (Worley, 1996; Gonzalez and Lowe, 2015).

The Worley Noise algorithm works by first deciding the size of the grid the cellular noise algorithm is going to be used on as well as how many Points the algorithm uses. Then these points will all be placed at random locations within the grid. In order to find the value at each pixel the distances between the pixel and each point will be calculated before the lowest value is returned as the height at the pixel (Gonzalez and Lowe, 2015).

When Worley Noise is output onto a 2d image texture it forms an image that looks similar to a field of cells.

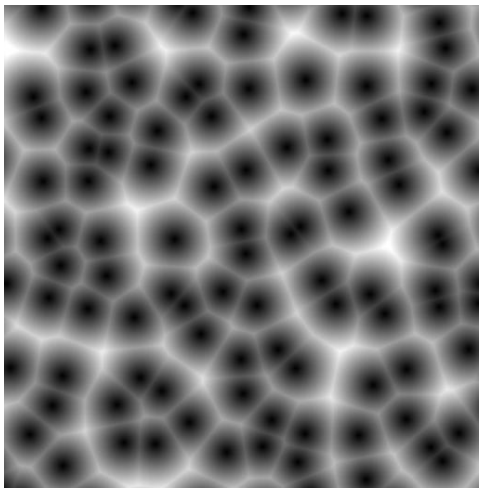

FIGURE 5 2D WORLEY NOISE TEXTURE (42 YEAH, 2023)

2.2 Literature Review

In this section relevant pieces of literature in the field of 3D terrain generation will be analysed and discussed to create a summary of existing knowledge and recent developments in the field of 3D terrain generation.

In a 2024 conference paper Jain, Sharma and Rajan (2024) proposed a frame that can be used in order to generate infinite 3D terrain. This method of terrain generation uses two deep learning modules. The first model is the terrain completion module which is trained to generate the infinite terrain which is then used by the second module which is known as the Terrain Enhancement module. This model is trained to enhance the detail of the terrain. Additionally, this module allows multiple different levels of detail for the terrain to be generated. This allows for the terrain to be more optimised in a gameplay setting where the terrain has to be rendered each frame, this is because the terrain that is far away from the player can be rendered at a low level of detail to improve performance while the terrain the player is nearest to can be rendered at the highest possible level of detail. Although this method for 3D terrain generation is impressive as it can generate terrain that looks very similar to terrain generated via the use of real world heightmaps, this paper does not fairly evaluate the terrain that is generated using their method with other advanced methods for terrain generation such as Fractal Perlin Noise instead only comparing it to the most basic forms of perlin noise based terrain generation.

In a 2023 journal article Huang and Yuan (2023) proposed a novel disentangled generative model named StyleTerrain using a Generative Adversarial Network (GAN) that can be used to generate controllable high-quality terrain. A Generative Adversarial Network is comprised of a generator network and a discriminator network (Huang and Yuan, 2023). The generator network learns and captures the potential distribution of the dataset and is tasked with generating new data (Gonog and Zhou, 2019). The role of the discriminator network is to attempt to determine whether the input data is from the actual dataset or if the data was created by the generator network (Gonog and Zhou, 2019). These two networks are trained through a zero-sum game where the discriminator attempts to correctly identify real and generated data and the generator attempts to trick the discriminator into incorrectly identifying the generated data as real data (Huang and Yuan, 2023). Due to one network's loss being another network's gain both networks must continually improve, resulting in increasingly realistic data generation (Gonog and Zhou, 2019). In the StyleTerrain created by Huang and Yuan (2023) the "Terra Advanced Spaceborne Thermal Emission and Reflection Radiometer (ASTER) Global Digital Elevation Model (GDEM) Version 3 (aka. ASTER GDEM V3)" (Huang and Yuan, 2023) data set was used as training data for their model. By using this dataset Huang and Yuan (2023) was able to produce highly realistic terrain heightmaps. One downside of this paper is that they only display a heightmap that they created by using an image. It would be greatly improved if this terrain was used on a 3D mesh as many papers do this (Pech, Lam and Masek, 2020; Golubev, Zagarskikh and Karsakov, 2016). Additionally other papers have described ways to increase the detail of terrain

scans such as Su et al. (2024)'s paper that details algorithms that enhance the detail of pre scanned DEM data. This DEM data is terrain scans of real-world terrain that are enhanced in this paper before being displayed in a 3D environment.

In 2020 an IEEE paper by Pech, Lam and Masek (2020) proposed a new novel method for generating terrain. This method was developed as a majority of research in the field of procedural terrain generation was focused on generating aesthetically pleasing landscapes with little attention given to generating terrain that could support gameplay mechanics and game design techniques. Pech, Lam and Masek (2020)'s terrain generation aimed to solve this problem by writing a terrain generation algorithm that includes game design features such as chokepoints, strongholds and hidden areas so that the terrain could be used to create maps for first person shooter (FPS) video games. This method allows game designers to have greater control on the outputted terrain allowing them to tailor the terrain to the type of map they would like to create. This allows this technique of terrain generation to be used on more games than just FPS games. The method proposed by Pech, Lam and Masek (2020) uses 3D Isovist values in order to generate terrain and in this paper they also proposed two new methods for accurately calculating the volume of an Isovist. Method 1 the "Isovist Method" (Pech, Lam and Masek, 2020) is ample to calculate the volume of a 3D Isovist by using a set of N Isovist radial vectors using the formula below

$$Volume = \frac{4\pi}{3}\frac{\sum_{n=1}^{N}|r_n|^3}{N}$$

FIGURE 6 THE ISOVIST METHOD FORMULA (PECH, LAM AND MASEK, 2020)

The second method for calculating the volume of a 3D Isovist that Pech, Lam and Masek (2020) proposed was a "Monte Carlo-Based" method. In this paper Pech, Lam and Masek (2020) states that Monte Carlo Integration is an established method of calculating the volume of an object but could not find an example of it being applied to calculating the volume of a 3D Isovist making this method a novel way to calculate the volume of a 3d Isovist. This method works by taking a set of random points from within the known volume of the Isovist and then using the points that are also inside the actual Isovist volume to approximate the size of the 3D Isovist. While this papers method of generating terrain generates terrain that both looks good as well as adheres to gameplay requirements from designers. One drawback of this method is that unlike perlin noise, the terrain generated in this method cannot be regenerated by using the same parameters. Due to this, this technique would not be suitable for generating terrain for vast open worlds seen it open world games such as Minecraft (Mojang, 2011) and No Mans Sky (Hello Games, 2016).

In a 2019 paper Liu et al. (2019) was able to use procedural content generation to be able to generate procedural levels that could be used in the strategy tower defence game 'Kingdom Rush: Frontiers' (Ironhide Game Studio, 2013) . In order to create these procedural levels, the main components of what makes up a level in 'Kingdom Rush: Frontiers' (Ironhide Game Studio, 2013) was first analysed in order to find the 3 main

building blocks that make up a level. This building blocks where Roads, Towers and Monsters. After breaking the levels down into these building blocks each of these building blocks was able to be generated procedurally using a variety of procedural content generation algorithms. A criticism of this paper is that the procedural generation algorithm that was created is only suitable for generating levels in one game and is not suited for use in other games. Although this is true, the technique of breaking down the levels into smaller building blocks and then creating algorithms to be able to generate each different building blocks could be applied to other genres of games. An example of how this could be used on another game is that in the game Overwatch (Blizzard Entertainment, 2016) you could break down the levels into the main building blocks of objectives, buildings and spawn locations. Additionally, many other tower defence games could be broken down into blocks that are very similar to the ones used in 'Kingdom Rush: Frontiers' (Ironhide Game Studio, 2013) for example the game Bloons Tower Defence 6 (Ninja Kiwi, 2018) could be broken down into Roads, Open Space and Balloons. Roads is the same as another one of the blocks used in 'Kingdom Rush: Frontiers' (Ironhide Game Studio, 2013) as well as Balloons that act as monsters in Bloons Tower Defence 6 (Ninja Kiwi, 2018). Meanwhile open space is similar to towers as the only difference is that in Bloons Tower Defence 6 (Ninja Kiwi, 2018) the towers can be placed anywhere where there is open space as appose to 'Kingdom Rush: Frontiers' (Ironhide Game Studio, 2013) that has set locations of the towers that can be built by the player.

In 2016 a paper by Golubev, Zagarskikh and Karsakov (2016) proposed a procedural terrain generation algorithm that used a modified version of the Dijkstra's pathfinding algorithm. This method allows for designers to have more control over the output of the terrain by modifying different parameters and weight functions. Although the terrain generated using this algorithm was highly controllable, the images of the mountain terrain shown in this method does not look very realistic or aesthetically pleasing. Due to this, this method is not suitable for generating mountainous terrain. Although one benefit of this method is that it generates incredibly realistic looking desert landscapes, other methods for terrain generations such as the methods proposed by Pech, Lam and Masek (2020) as well as perlin noise-based landscapes are not used to generate dessert terrain which means that this PCG technique fills a gap in knowledge of generating desert terrain.

Chapter 3: Design

3.1 Terrain

The terrain designed in this game will be rocky mountainous terrain with paths for a skier to ski down. These paths will be created with Unreal Engines Terrain spline paths. These paths will also be generated procedurally by using random points on the terrain. Additionally, these paths should also go downwards, and the finish should be at the bottom of the mountain.

The rock parts of the terrain will be generated by using a terrain height map. These height maps should have noise functions applied to them. A fractal perlin noise function will be used to create detailed rocky terrain, and a Worley noise algorithm will be created to add ridges to this terrain for a more realistic look.

Additionally, the material used on the mountain should be a snowy material that becomes a rocky material when the slope is very steep.

3.2 Gameplay

The main gameplay loop of the game is to try and reach the bottom of the mountain as fast as possible, a Game timer will be used to track how long it is taking the player to complete the level and the player will be incentivised to replay the level to take advantage of the extra replay ability that is gained from using procedural content generation.

Additionally, checkpoints will be positioned along this track and all of the checkpoints must be passed for the player to complete the level.

3.3 Character Controller

The character controller will be physics based and be controlled by applying forces to the player. To move forward the player will be able to push the ski poles along the ground to gain a boost of speed. Additionally, when the player is moving at high speeds the skier should lean down to become aerodynamic and move faster because of this. This is to make the games movement more realistic as it would be hard for a skier to push themselves along with their poles if they're already moving at a high speed. The player must also be able to slow themselves by breaking, this will trigger an animation and slow the player down. Finally, the player must also be able to steer left and right.

3.4 Skier Character

The player character model will be in a low poly art style as it will be based off the player model from Loney Mountains: Snow Riders (Medagon Industries, 2024). An image of this characer is shown below.

FIGURE 7 LONELY MOUNTAINS SNOW RIDERS (MEDAGON INDUSTRIES, 2024)

It should include a helmet as well as ski poles, goggles and skis. Additionally, it should include the following animations:

- Standing Still (Idle)
- Pushing (Move forward)
- Leaning Forward
- Breaking
- Falling

3.5 User Interface (UI)

A main menu will be used so that the player can start and quit the game. A design was created of the main menu and is displayed in the image below. The Menu on the left-hand side will be created with a user interface and the grey box that says skier character will be the 3d model of the skier. Finally, a mountain background will be created with Unreal Engines Landscaping tools.

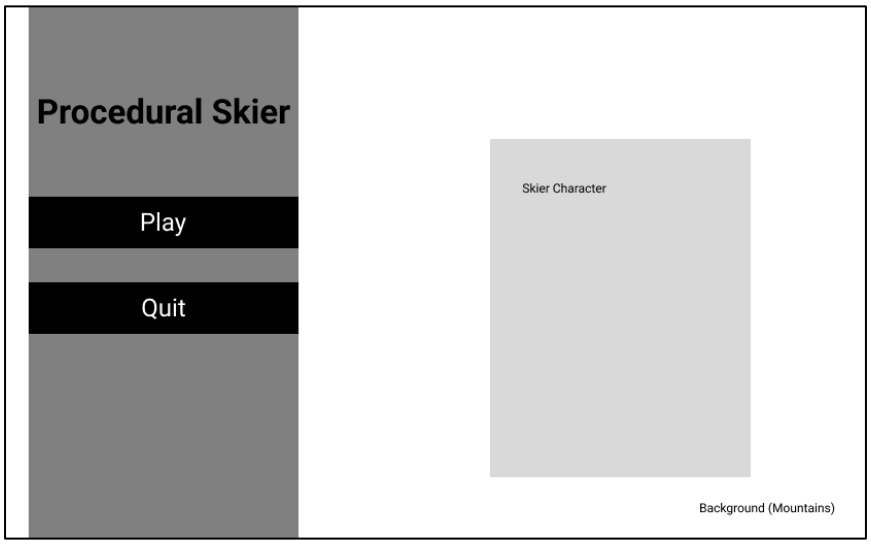

FIGURE 8 MAIN MENU DESIGN

A design for the level complete menu was also created. It features the time that the user completed the level in as well as an option to restart the level and return back to the main menu.

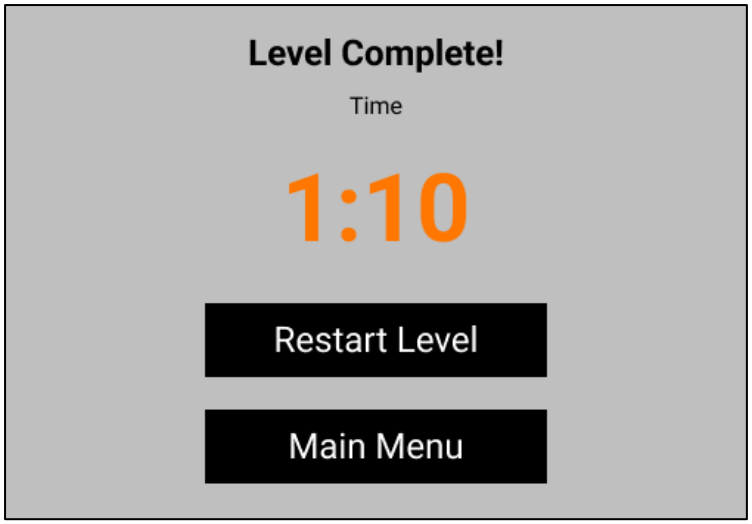

FIGURE 9 LEVEL COMPLETE DESIGN

The game will also feature text at the top of the screen that displays a timer of the current level so the user can keep track of their progress while they are completing the level.

Chapter 4: Development

4.1 Terrain

The first step in creating the terrain was to create a flat landscape. The ALandscape component was used for this as it can generated terrain based on an unsigned 16-bit integer heightmap as well as the landscape splines to shape paths into the terrain. To create this terrain a C++ class known as TerrainGenerator.cpp as created as well as a header file named TerrainGenerator.h. The function below was created to gnerate a flat landscape. In order to create a landscape Quads Per Component and SizeX and SizeY must first be calculated. The heightmap that is used in this flat terrain has every vertex set to 32768 as this is the middle value of the uint16 variable.

```
TArray<FLandscapeImportLayerInfo>MaterialImportLayers;
MaterialImportLayers.Reserve( Number: 0);

TMap<FGuid, TArray<uint16>> HeightDataPerLayers;
TMap<FGuid, TArray<FLandscapeImportLayerInfo>> MaterialLayerDataPerLayers;

// Calculate QuadsPerComponent, SizeX and SizeY
int32 QuadsPerComponent = SectionSize  *  SectionSize;
int32 SizeX = ComponentCountX * QuadsPerComponent + 1;
int32 SizeY = ComponentCountY * QuadsPerComponent + 1;

TArray<uint16> HeightMap;
HeightMap.SetNum(SizeX * SizeY);
for (int32 i = 0; i < HeightMap.Num(); i++)
{
    HeightMap[i] = 32768;
}

HeightDataPerLayers.Add( InKey: FGuid(),  InValue: MoveTemp(HeightMap));
MaterialLayerDataPerLayers.Add( InKey: FGuid(),  InValue: MoveTemp(MaterialImportLayers));

ALandscape* Landscape = GetWorld()->SpawnActor<ALandscape>();

Landscape->Import(FGuid::NewGuid(),  InMinX: 0,  InMinY: 0,  InMaxX: SizeX  -1,  InMaxY: SizeY -1 ,
    InNumSubsections: SectionsPerComponent, QuadsPerComponent, HeightDataPerLayers,
    InHeightmapFileName: nullptr, MaterialLayerDataPerLayers, ELandscapeImportAlphamapType::Additive);
```

FIGURE 10 FLAT TERRAIN FUNCTION

4.1.1 Perlin Noise

The Perlin Noise Function is a commonly used function for procedural content generation. Due to this, most game engines include an in-built function that allows the use of Perlin Noise. In Unreal Engine the function FMath::PerlinNoise2D is available to be used (Epic Games, N.d). This function returns a perlin noise sample at a Vector 2D location(Epic Games, N.d). This value will range from -1.0 and 1.0 (Epic Games, N.d). As the perlin noise function is a relatively complex noise function that would require a pseudo random number function to also be written the usage of the inbuilt noise

function will be opted for in this case. Another library that was considered before opting for the in-built Unreal function was an Unreal Engine Plugin called Fast Noise Generator (Rockam, 2020) which allows access to the fast Noise Library in Unreal (Rockam, 2020; Peck, 2024), the inbuilt function was opted for in this case as it can be simply just used in Unreal Engine and does not require any extra dependencies or installations.

The perlin noise function will be applied to the terrain by looping through each vertex on the terrains heightmap, using the X and Y coordinate of this vertex and an input and adding its output on top of the default value for the flat terrain (32768). Additionally, before Octaves were added to this perlin noise the X and Y coordinate Vector was multiplied by a variable called Frequency. This variable will control the scale of the noise, a higher frequency will cause faster changes in height while a smaller frequency will cause quicker changes in height (Etherington, 2022). Finally, before being added onto the height map the perlin value returned by the function was multiplied by another variable called amplitude. This variable controls the maximum and minimum value that can be added to the terrain by the perlin noise function. A larger amplitude will cause the terrain to have a greater variance in height and a smaller value will cause the effect of the perlin noise function on the terrain to be less noticeable, due to the smaller variance in height (Etherington, 2022). An image of the terrain with perlin noise applied to it is shown below. This image uses terrain with an amplitude value of 1700.0 and a frequency value of 0.015.

FIGURE 11 PERLIN NOISE TERRAIN OUTPUT

As shown in the image above this function creates smooth rolling hills on the terrain.

4.1.2 Fractal Perlin Noise

In order to enhance the detail of the Perlin Noise terrain multiple layers of this noise will be added on top of each other. This is known as adding octaves of fractal perlin noise (Etherington, 2022; Lague, 2016). Each one of these octaves of noise will be added with a higher frequency and a lower amplitude then the previous octave (Lague, 2016). This will create a rougher and rockier look on the terrain. In order to add octaves of perlin noise 3 new parameters will be added to the code. These parameters will be, Octaves,

this parameter will control the number of octaves of perlin noise added to the terrain. Lacunarity, this will control how much the increase in frequency for each octave (Lague, 2016), and finally Persistence which will control how much the decrease in amplitude is for each octave (Lague, 2016).

Due to these new parameters the octave frequency and octave amplitude must be calculated before using the perlin noise function. The following formulas are used to find the octave frequency and octave amplitude.

$$Octave\ Frequncy = Freqency\ of\ first\ octave * (Lacanarity * Octave\ Number)$$

FIGURE 12 FREQUENCY FORMULA

$$Octave\ Amplitude = \frac{Amplitude\ of\ first\ octave}{(Persistance * Octave\ Number)}$$

FIGURE 13 OCTAVE AMPLITUDE FORMULA

After the new values for frequency and amplitude have been calculated the value of the octave can then be calculated in the same way as in the perlin noise without octaves. Expect using Octave Frequency and Octave Amplitude in place of the standard Frequency and Amplitude values. This will be looped for each octave in defined by the octave value and the perlin values will be added onto the height map each time in order to have the sum of all the octaves added together at the end. An example of terrain generated using this method is shown in the screenshot below. The landscape in this screenshot uses the same values for amplitude and frequency as the perlin noise terrain that does not include octaves that is shown in the previous image in this report. Additionally, the values of the new parameters are an octave count of 5, a persistence of 4.0 and a lacunarity of 2.0.

FIGURE 14 FRACTAL PERLIN NOISE TERRAIN OUTPUT

As shown in this image fractal perlin noise adds a lot of detail to the train and makes it much rougher and rockier.

4.1.3 Worley Noise

The first step when adding Worley Noise to the terrain was to decide what Worley Noise function will be used if there is one available or whether it would be better to write one manually. As there is no Worley Noise function in-built into Unreal Engine an in-built function cannot be used when implementing Worley Noise in the project. There were many different approaches that could have been chosen for the method of implementing Worley Noise, one method that could have been used was the free Unreal Engine Plugin by Rockam (2020) called Fast Noise Generator, this plug in allows access to the Fast Noise library (Peck, 2024) in Unreal Engine (Rockam, 2020) which is a library that has a large collection of different noise algorithms in it such as Value Noise, Perlin Noise and Worley Noise. Although this plug in has many good arguments for using it over manually programming a Worley Noise Function, the manual programming option was chosen to complete this task. This was chosen as manually programming the noise function would give greater control over the code that is being run in the project and allow the function to be fine-tuned to match the projects needs and for the function to be better integrated with the rest of the project.

When implementing Worley Noise two functions where created. The first function is called Initialize Worley Points. This function adds X number of random Vector 2d points that fit withing the coordinates of the terrain to an array called WorleyPoints. This method of generating points has been chosen as the points do not need to be regenerated every time the function is called unlike alternative methods where the coordinates of the points are generated each time with a Pseudo Random Number Generator. This will cause this Worley Noise function to be more optimised than other methods. It is also important to note that this method of generating the points first could be used to generate the rotator vectors first when implementing perlin noise but this method is generally not chosen for this as when generating perlin noise there is much more rotator vectors to store than the number of points needed for a Worley Noise Algorithm. Additionally, this does not need to be used as the terrain in this project does not need to be infinitely tile able.

The next step for creating a Worley Noise function is to create the actual function that will be called for each vertex in the terrain mesh. This function should return a float that represents the height that that should be added onto that point. The name of this function matches the naming convention used in the in-built function for perlin noise which is called PerlinNoise2D, Due to this the name for this function will be WorleyNoise2D and like the Perlin Noise function require an X and Y coordinate to be input. This function first works by creating a float value called MinDistance. This value will be equal to FLT_MAX which is the largest possible float value assignable. After this variable has been created, each point will be looped though the distance between that point and the coordinate will be calculated. This calculation will use the FVector2D::Distance function but could also be calculated using the Euclidean Distance Formula where (x_1, y_1) = the coordinate of the vertex and (x_2, y_2) = the coordinate of the point (Chugani, 2024). This function is shown below.

$$Distance = \sqrt{(x_2 - x_1)^2 + (y_2 + y_1)^2}$$

FIGURE 15 2D EUCLIDEAN DISTANCE FUNCTION (CHUGANI, 2024)

Ultimately, the FVector2D::Distance function was chosen as it is a relatively simple function that is in built into Unreal Engine, so a function does not need to be manually written. After the distance has been calculated it will be compared with the value stored in the MinDistance variable. If it is smaller than the MinDistance variable, then the MinDistance variable will be updated to this new value. Once the loop has been completed the distance between the nearest point and the input coordinate will be stored in the MinDistance variable so it will simply be returned as the noise value at that coordinate (Gonzalez and Lowe, 2015). The value returned from this function will be multiplied by a variable called Worley Amplitude before being added onto the height of the vertex. This multiplication will cause the effects of the Worley noise function on the landscape to be more exaggerated. Below is an image of the output of this function on a landscape mesh. This function creates ridges in the landscape much like real mountains.

FIGURE 16 WORLEY NOISE LANDSCAPE OUTPUT

4.1.4 Combining Fractal Perlin and Worley Noise

The final algorithm of perlin noise that will be used in this project will create a heightmap based terrain via a combination of both the fractal Perlin Noise Algorithm and the Worley noise algorithm. This was done by adding both the results from the fractal perlin noise algorithm and the Worley noise algorithm together onto the same height map. The resulting image is an image that had the rocky hills that are created using the fractal perlin noise algorithm but with the ridges generated via the Worley noise algorithm added on top of them. This creates a highly detailed and realistic mountain landscape. An image of the terrain created with this method is shown below.

FIGURE 17 COMBINATION OF FRACTAL NOISE TERRAIN AND WORLEY NOISE TERRAIN

4.1.5 Spline Terrain

The next step was to carve the terrain by using splines. This would function as paths that the player could ski down. For this a Add Spline Function was created that took a Array of Vector3 spline point locations as well as the ALandscape component. This function would add splines by first adding a spline to the terrain and then looping through each spline point and adding it on to this spline. This function did not work as intended as after the spline points were added to the terrain. The function to refresh the terrain so that it deforms around this spline was not accessible via C++. Although, this function can be used when creating a terrain manually instead of procedurally. Below is the code that was written to be able to add splines to this terrain but with a comment in place of where the update landscape function should be.

```
62
63  void AddSpline(ALandscape* Landscape, TArray<FVector>& SplinePoints)
64  {
65      if (!Landscape || !Landscape->GetSplinesComponent())
66      {
67          return;
68      }
69
70      ULandscapeSplinesComponent* SplineComponent = Landscape->GetSplinesComponent();
71      ULandscapeSplineControlPoint* PreviousControlPoint = nullptr;
72
73      for (const FVector& Point : SplinePoints)
74      {
75          ULandscapeSplineControlPoint* ControlPoint = NewObject<ULandscapeSplineControlPoint>(SplineComponent);
76          ControlPoint->Location = Point;
77          SplineComponent->GetControlPoints().Add(ControlPoint);
78
79          if (PreviousControlPoint)
80          {
81              ULandscapeSplineSegment* SplineSegment = NewObject<ULandscapeSplineSegment>(SplineComponent);
82              SplineSegment->Connections[0].ControlPoint = PreviousControlPoint;
83              SplineSegment->Connections[0].TangentLen = 0;
84
85              SplineSegment->Connections[1].ControlPoint = ControlPoint;
86              SplineSegment->Connections[1].TangentLen = 0;
87
88              SplineComponent->GetSegments().Add(SplineSegment);
89          }
90
91          PreviousControlPoint = ControlPoint;
92      }
93      // Update Landscape Function Should Go Here
94  }
```

FIGURE 18 ADD SPLINE FUNCTION (NOT WORKING) (NOTE THIS FUNCTION HAS NOT BEEN INCLUDED IN THE ARTEFACT)

Due to the splines not working as intended in the plan a different plan for the gameplay had to be adopted. This will be discussed further in the Checkpoint section of this chapter.

4.1.6 Terrain Material

The material that is applied to the terrain uses a free to use snow texture and rock textured that was downloaded from Poly Haven and created by (Tuytel, bN.d; aN.d).

Additionally, the World Aligned Blend node with a Blender Sharpness of 25.0 and a Blend Bias of -10 was used as the alpha in a lerp node within the blueprint materials editor in order to create a material that should blend between showing the rock texture on steep faces and the snow texture on relatively gentle slopes.

4.2 Skier Character Model

The 3D model for the player character was created using blender 4.0. This was chosen as it is a free 3D modelling software that is powerful enough to create the models needed in this project. Due to the low poly style of this 3d model it was created by starting with a basic cube, mirroring it and then extruding and insetting the shape to create a skier. Below is an image of the initial skier model before it was rigged and textured and before props such as the goggles and skis have been added to the character.

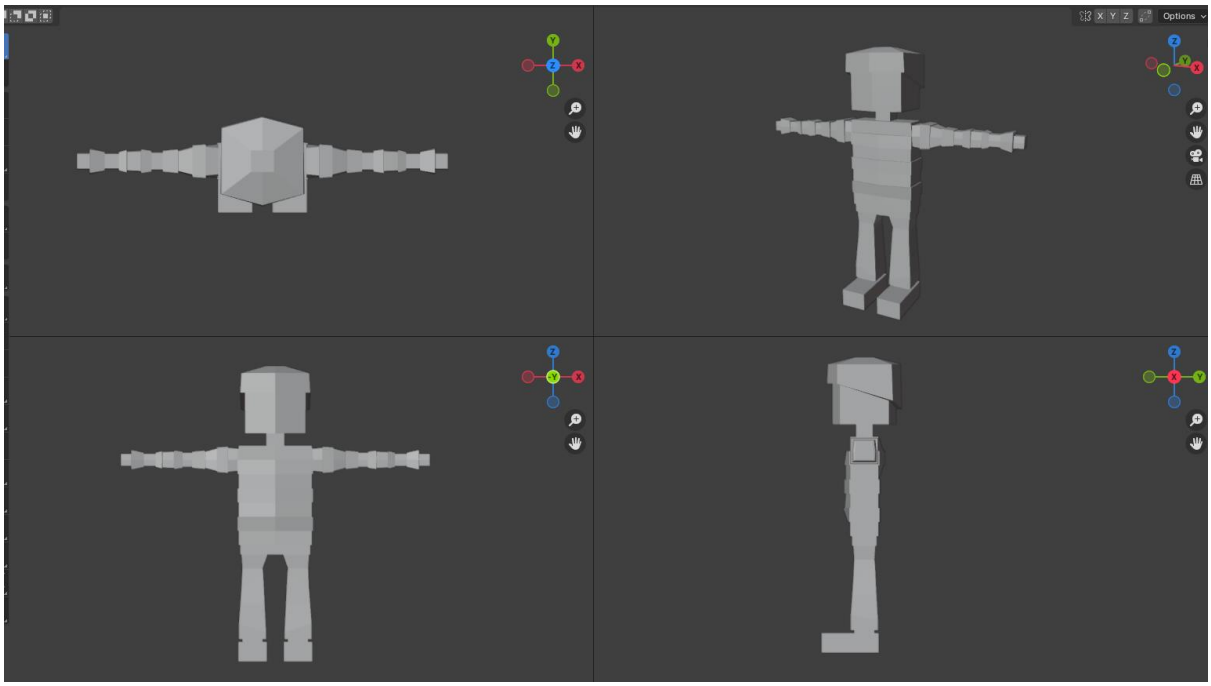

FIGURE 19 INITIAL SKIER MODEL

4.2.1 UV Mapping

After the initial model for the skier was created, the model was UV unwrapped in order to create a UV map of the model. A UV map is a 2d representation of the surface of a 3D model (Denham, N.d). This will allow textures to be applied to the model without them being stretched. Below is firstly an image of the model with the seems in red and secondly an image of the UV map generated when this model is unwrapped with those seems.

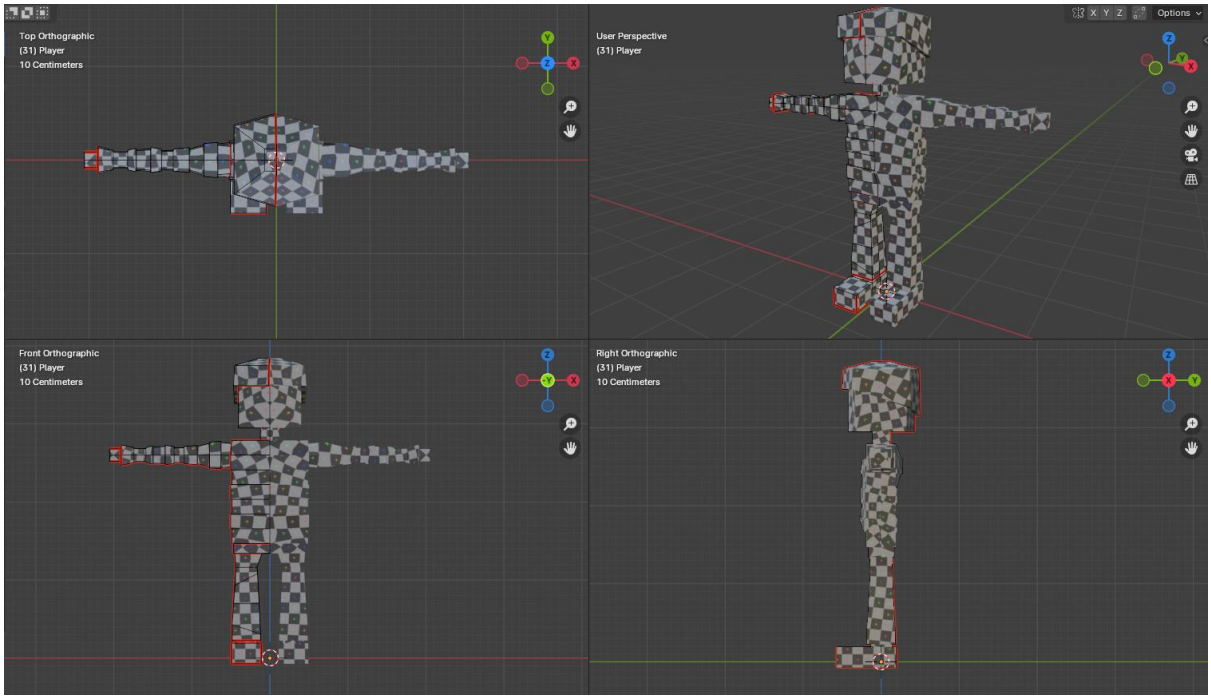

FIGURE 20 SKIER MODEL WITH SEAMS

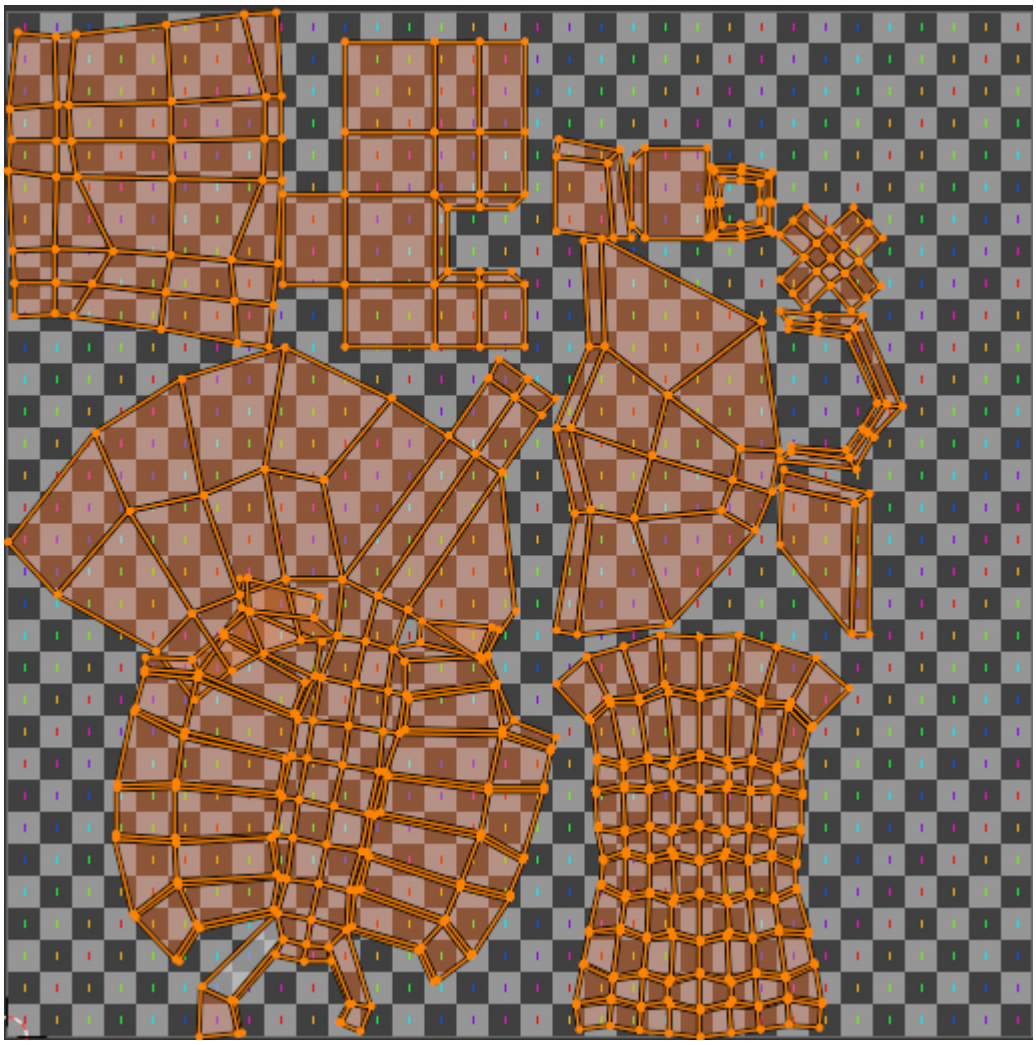

FIGURE 21 SKIER MODEL UV MAP

4.2.2 Textures

Once this UV Map was generated, textures were applied to the skier. Due to the low poly style of the skier the fill tool was used in order to texture paint solid colours onto different parts of the Mesh. Below is an image of firstly the 3D model with the texture applied to it and secondly an image of the texture created via texture painting.

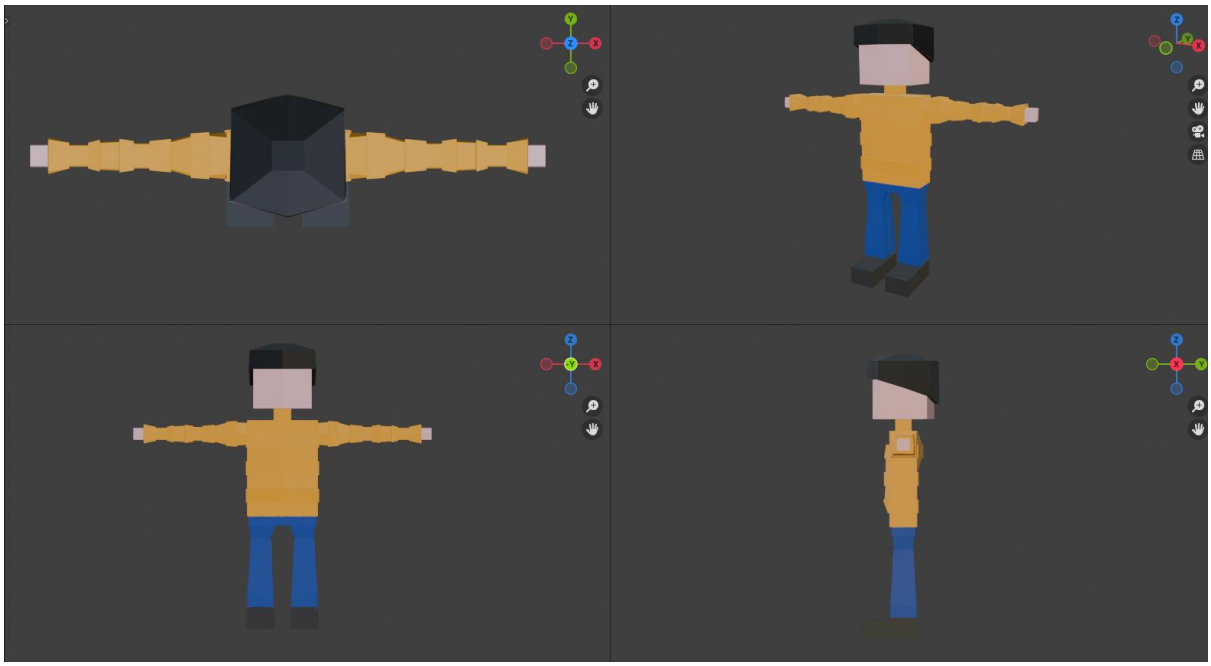

FIGURE 22 SKIER MODEL WITH TEXTURES

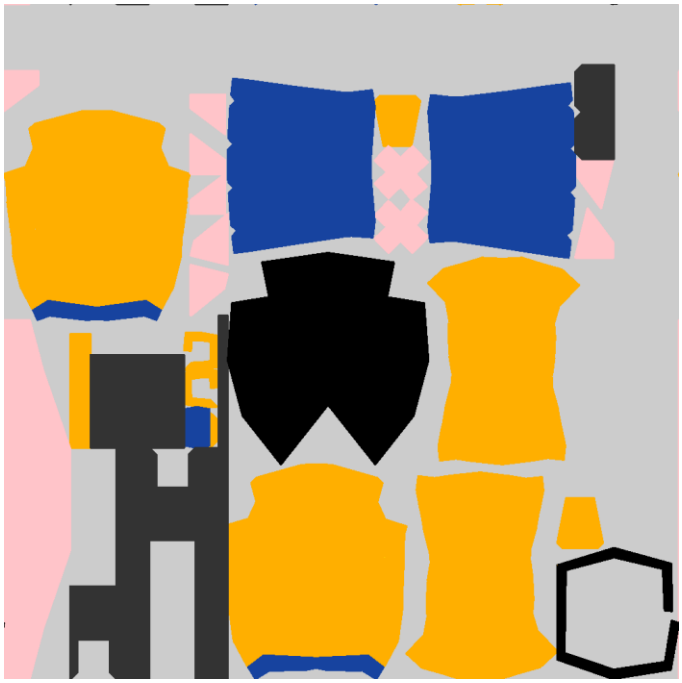

FIGURE 23 SKIER MODEL TEXTURE

4.2.3 Rigging

Once the skier was modelled and textured the next step was to rig the model so that it is prepped for animations to be created with it. Initially the plugin Rigify was used to

create a rig for the skier mesh. This was chosen to be used as it is a very fast way to create a rig for a humanoid character that includes a full functioning IK (inverse kinematics) rig. This method worked perfectly in blender and some animations were created for the rig before they were imported into Unreal Engine but when they were imported into Unreal Engine there was major scaling and clipping issues with the animation. To test that this wasn't a problem with the import into Unreal Engine, Autodesk's FBX review software was used to see that the exported animations looked like outside of both Blender and Unreal Engine. Below is an image of what the issues with the exported animations look like.

FIGURE 24 SKIER BROKEN ANIMATIONS IN FBX REVIEW

As shown in the image above, the animations issues where still present in FBX review. Due to this the rig and animations created via the Rigify plugin was scrapped and a new rig was created. This new rig was created without the use of any plugins but a YouTube video tutorial created by Richstubbsanimation (2023) was used to gain the knowledge necessary to create an IK rig. The IK rig created for the skier character includes a deform rig that deforms the skiers actual mesh for the animations. A driver rig that this deform rig copies as well as a few extra bones were added as a control rig. These extra bones include an extra bone at the hip to allow for the legs and arms to bend when the hip is moved as well as IK bones at the hands and feet, as well as pole vectors for the knees.

Additionally, these bones had there meshes changed so that they would stand out as part of the control rig and make the animation process is more intuitive.

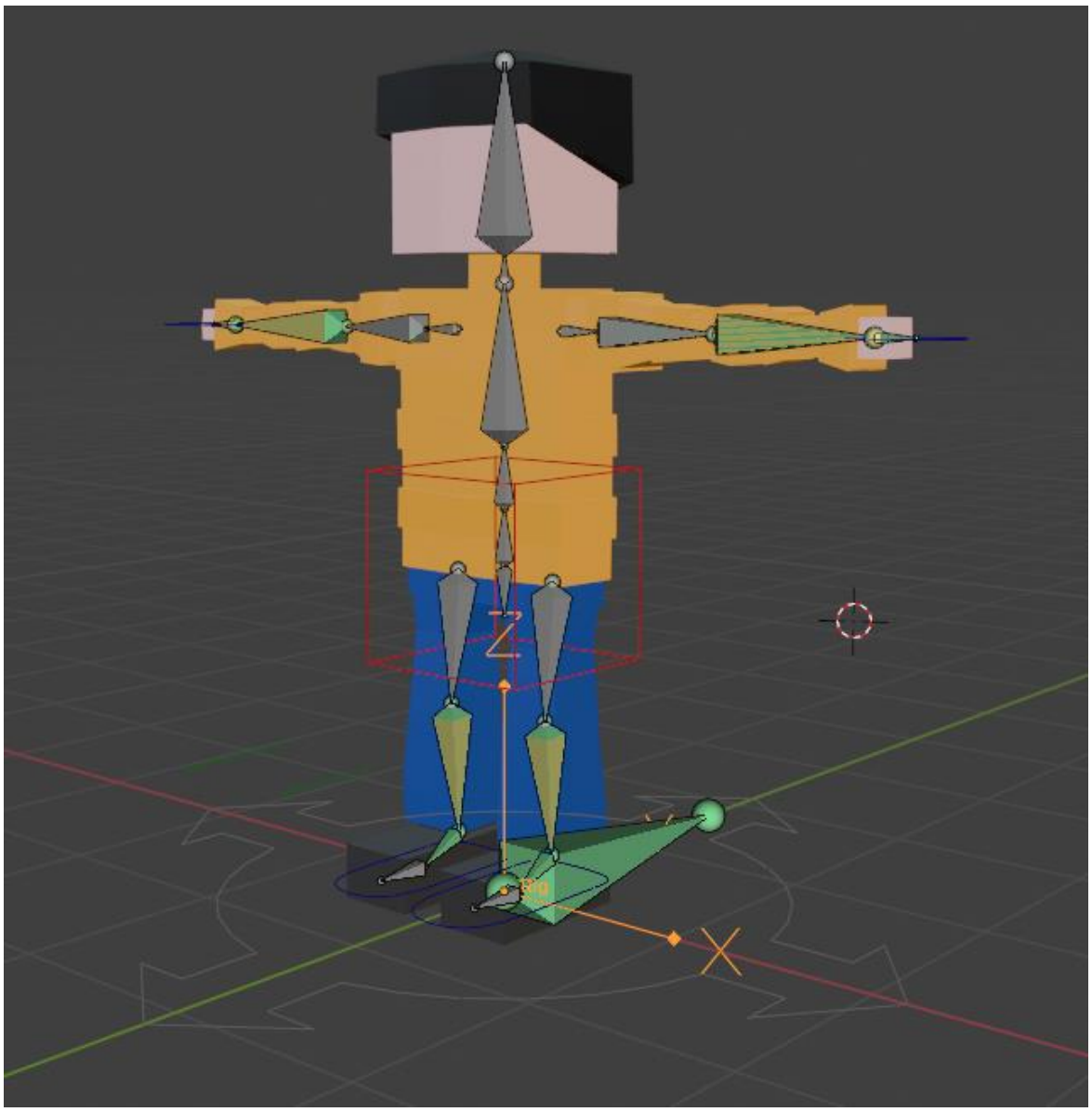

FIGURE 25 SKIER RIG

4.2.4 Animations

As discussed previously during the design process 5 animations where identified as needing to be made. These animations where a idle animation, a pushing forward animation, a leaning forward animation, a breaking animation and a falling animation. As shown in the below image all 5 of these animations where created.

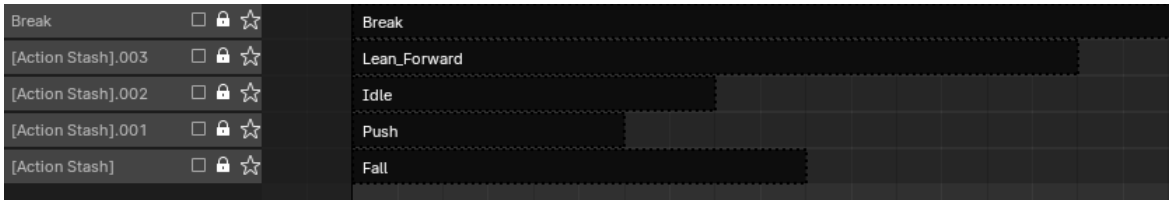

FIGURE 26 SKIER ANIMATION LIST

4.2.5 Props

The final step in creating the skier model is to create the props. These props include 2 skier poles to be added to the skier's hand, 2 skiers to be added to the skier's feet and a pair of goggles to be added to the player's head. Both the poles and skis were created on one side of the skier and then mirrored to the other side so that they did not have to be made twice. All of these models have been attached to the rig by parenting them to the bones at the location required and the same method was used for texturing as was used to UV map and texture the main mesh of the skier model.

Below is an image of the ski goggles created for the skier. These were created by first extruding and scaling a 2D plane to create the front of the goggles. Then parts where extruded outwards to add depth to the goggles.

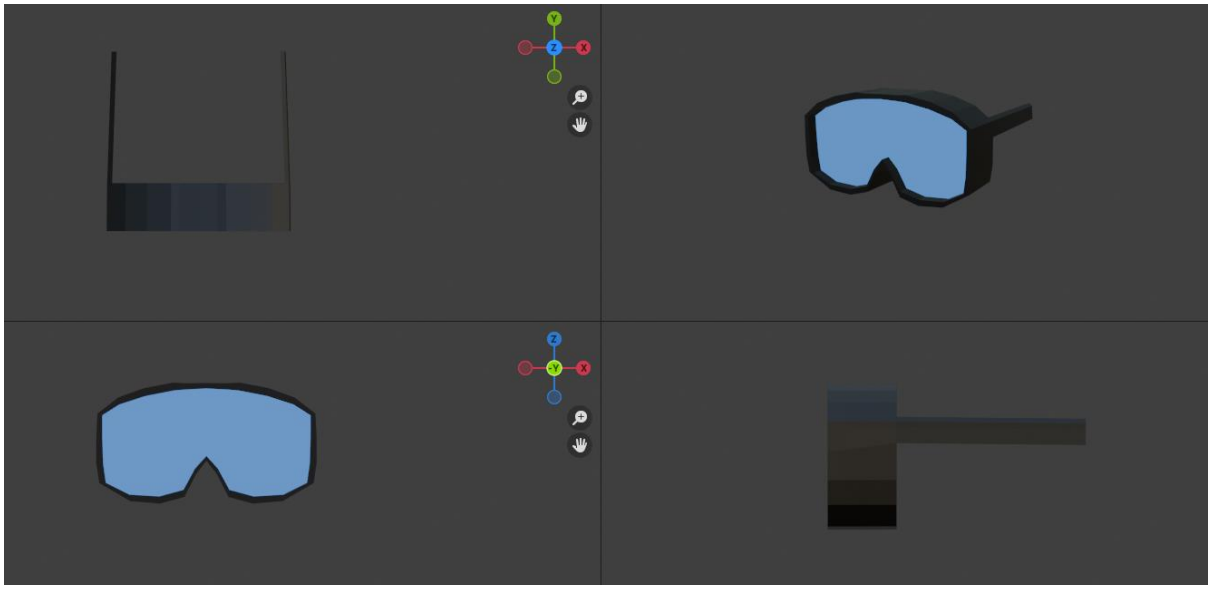

FIGURE 27 SKI GOGGLES

Next the ski poles were modelled. This was modelled by first spawning a 6-sided cylinder then this cylinder was scaled to the height required and loop cuts were used to make the black sections of the poles wider. Additionally, the handle of the poles were created by extruding out the top of the cylinder, increasing the height of this extrusion, scaling and then slightly rotating the extruded face. Finally, the top and bottom of the poles were edited so that they come to a point instead of being a flat face. This does require the use of tris to be able to achieve this look. An image of the ski pole model is shown below.

FIGURE 28 SKI POLES 3D MODEL

The final prop created for this model was the skis that the skier is standing on. These where created very similarly to how the goggles where created. First the surface at the skier stands on was modelled using a 2D plane then it was given thickness by extruding downwards to create the bottom of the skis before scaling this to be smaller than the top. An image of these skis is shown below.

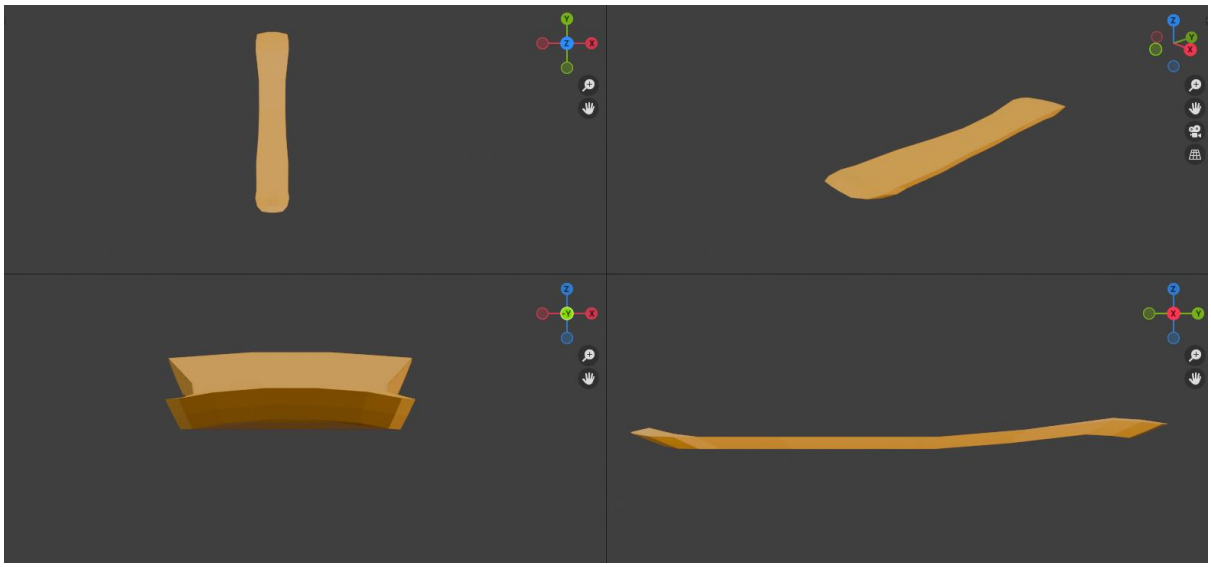

FIGURE 29 SKIS MODEL

4.2.6 Exporting

Finally, the skier's mesh, animations and texture were exported from blender as an FBX file. Tested in Autodesk FBX review and then imported into Unreal Engine. Below is an image of the skier in Unreal Engine with working animations and textures.

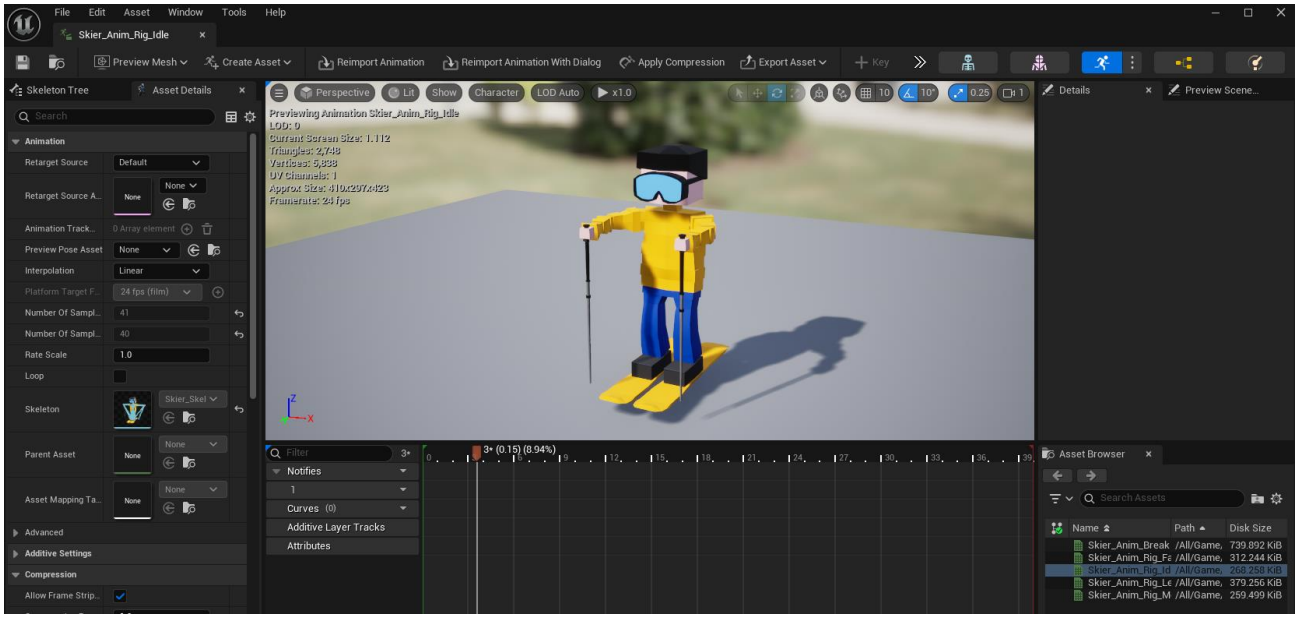

FIGURE 30 IMAGE OF SKIER MODEL EXPORTED INTO UNREAL ENGINE

4.3 Character Controller

When creating the character controller for the skier it was initially created by using Unreal Engines Character Class in C++. This class had many built-in functions that help when creating a character controller, but the character controller created using this class was ultimately scrapped as it didn't allow for the character to be solely controlled by physics and forces like a static mesh object. Due to this the pawn class was adopted as the class for the player character. This character uses 4 different Unreal Engine components. These components are a Capsule Collider that will be used as a parent to the other components in the class. This component will also be the component that has forces applied to it for the skier to be able to move. The next component is the Skeletal Mesh Component this will hold the Mesh and animations created in the previous section of this report, a Spring arm component that will place the camera in 3rd person as well as the camera component that allows the scene to be viewed by the player.

The main 3 actions the player is able to do in the character controller is move forward (either with a push or by leaning), Steer (Either in one place or by leaning) and by breaking

A variable called Forward Input will keep track of weather the player is trying to move forward. When the player clicks the W key this variable will be set to a positive value and when the player releases this key it will be set to a negative value. If the player is trying to move forward and is on the ground, there will be a check for what the current speed of the player is. If it is below the max push speed the skier will start the animation

to push forward. The forward velocity gain by doing this animation is triggered using an animation notify in the skier animation. This is so that the forward force can be applied at the correct time in the animation. If the player is above this speed threshold, then the player will start the leaning animation. While the player is leaning the physics material on the player will change to a material that has a lower friction. This will cause the player to maintain their speed for longer, additionally a small forward force will be added at all times when the player is leaning. This will cause the player to increase speed if they are leaning downhill. Additionally, this will be multiplied by delta time to stop the leaning speed increase from changing depending on frame rate. Unlike the leaning force the pushing force does not need to be multiplied by delta time as it is only activated on one frame.

When the player is holding the A or D key the player will steer left and right. If the player is moving at a low speed than the player will rotate on a point but if the player is moving at a higher speed a force will be added to either side of the player in order to make them move in a way that simulates leaning. Additionally, this sideways force will be multiplied by a float curve based on their current speed. This will cause the player to rotate fast when the player is either moving fast or moving slowly but taper off to a slow turning speed in the middle. Below is an image of the turning curve.

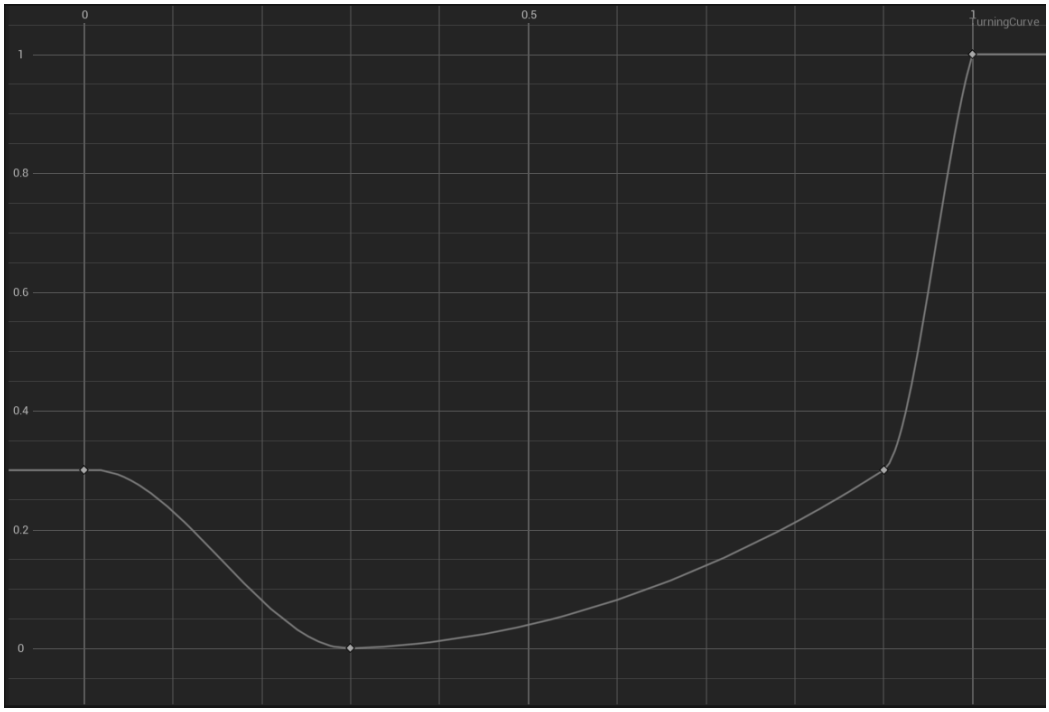

FIGURE 31 SKIER TURNING CURVE

A backwards force will be added to the player if they click the S key. Like the pushing forward forces this will be applied to the player via the use of animation notifies. This is because the force should be added to the player each frame they are in this animation. Additionally, this force will be multiplied by delta time to avoid the breaking speed changing depending on frame rate.

4.3.1 Ground Check

As the pawn class is being used instead of the character class, a custom function must be written to check if the player is grounded or not. This function uses a variable called ground check distance and casts a trace downwards to see if the player is standing on the ground. This check distance allows for leeway so that the player can do actions that require them to be grounded when they are nearly on the ground. Additionally, the players collision is ignored during this RayCast as if it was not the ray would always collide with the player. Below is a screenshot of this function.

```
151  bool ASkier_Character::GroundCheck(FVector PlayerLocation)
152  {
153      FVector EndLocation = PlayerLocation - FVector( InX: 0,  InY: 0,  InZ: GroundCheckDistance);
154
155      FHitResult Hit;
156      FCollisionQueryParams Params;
157
158      Params.AddIgnoredActor(this);
159
160      bool grounded = GetWorld()->LineTraceSingleByChannel(
161          [&] Hit,
162          Start: PlayerLocation,
163          EndLocation,
164          TraceChannel: ECC_WorldStatic,
165          Params);
166
167      // Draw Debug Line
168      //DrawDebugLine(GetWorld(), PlayerLocation, EndLocation, grounded ? FColor::Green :
169
170      return grounded;
171  }
```

FIGURE 32 GROUND CHECK FUNCTION

4.3.2 Animation Control

The animations in this project are controlled via an animation montage inside of an animation blueprint. This montage uses the Boolean variables Moving Forward (Leaning), Is grounded, Forward Input, Pushing and breaking in order to pick which animation is chosen to be played.

Below is a screen shot of this animation montage.

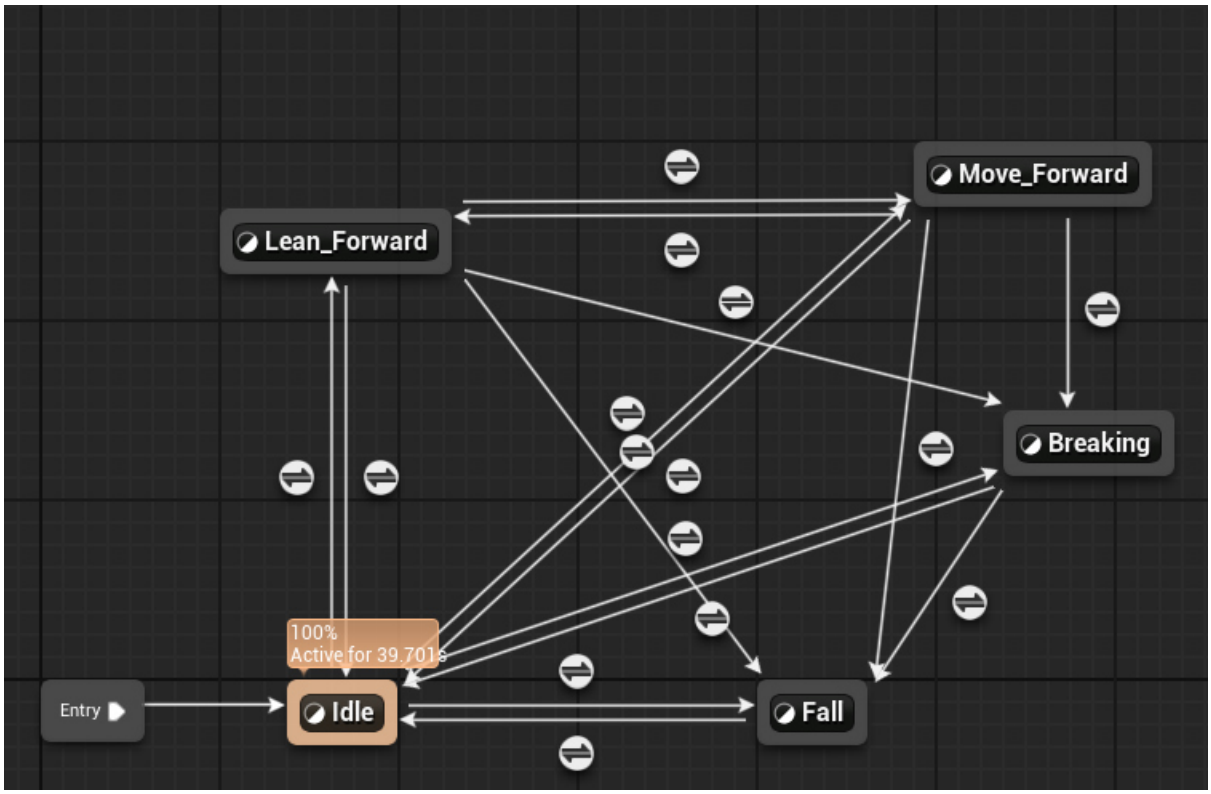

FIGURE 33 SKIER ANIMATION MONTAGE

4.4 Checkpoints

In the design chapter of this report the checkpoints in this game were planned to be like checkpoints in a standard racing game. They would appear along a spline path that has been created during terrain generation. The player would have to race between them in order until they reach the finish line, and their score would be determined by the time it took them but during the implementation of the splines onto the terrain there was an issue that caused the spline to not be able to be added. Due to this an alternative design for the gameplay would have to be created.

The new plan that was decided on was for checkpoints to appear all around the terrain and the player has to collect as much of them as possible before a timer runs out. Their score would equal the number of checkpoints they have collected.

In order to create these checkpoints, the maximum size for the terrain must first be calculated. This has to be calculated as the X and Y values used in the terrain generation refer to what the coordinate of the vertex is in the grid e.g. the second vertex has a coordinate of (1,0) but the checkpoints need to spawn in world space. To do this

this formula is used where Size X is the same Size variable used to generate the terrain grid. As well as the equivalent formular to calculate Max Y where Size Y is used instead of Size X.

$$Max\ X = (SizeX - 1) \times 128$$

FIGURE 34 MAX X FORMULA

Once this formula is calculated a random checkpoint coordinate is generated using FMath::RandRange and setting the range between 200 and the maximum coordinate - 200. 200 was used as the start value and taken away from the maximum value as it prevents the checkpoints from being spawned on the edge of the terrain. Once this coordinate is calculated the Z value of the checkpoint is calculated by performing a downwards RayCast on the terrain and finding the height were the ray hit the terrain. Next a random Y rotation for the checkpoint is generated and the checkpoint is spawned using the spawn actor function.

This checkpoint has a USphere component that is being used as a trigger. When the player enters the trigger then the checkpoint will generate another checkpoint into the world, destroy itself and call a function on the levels game mode that will increase the score and update the UI that displays the score.

Additionally, the checkpoint uses a 3D model that was created using a primitive cylinder and extruding and merging edges in blender and is shown in the screen shot below.

4.5 UI

All of the UI in this project was created using Widget Blueprints. Both the widget for the heads-up display and the end level screen uses Blueprint Implementable Events that are called via C++ in the game mode class.

The first blueprint that was created was the main heads-up display (HUD) this HUD displays the time the remaining in the level.

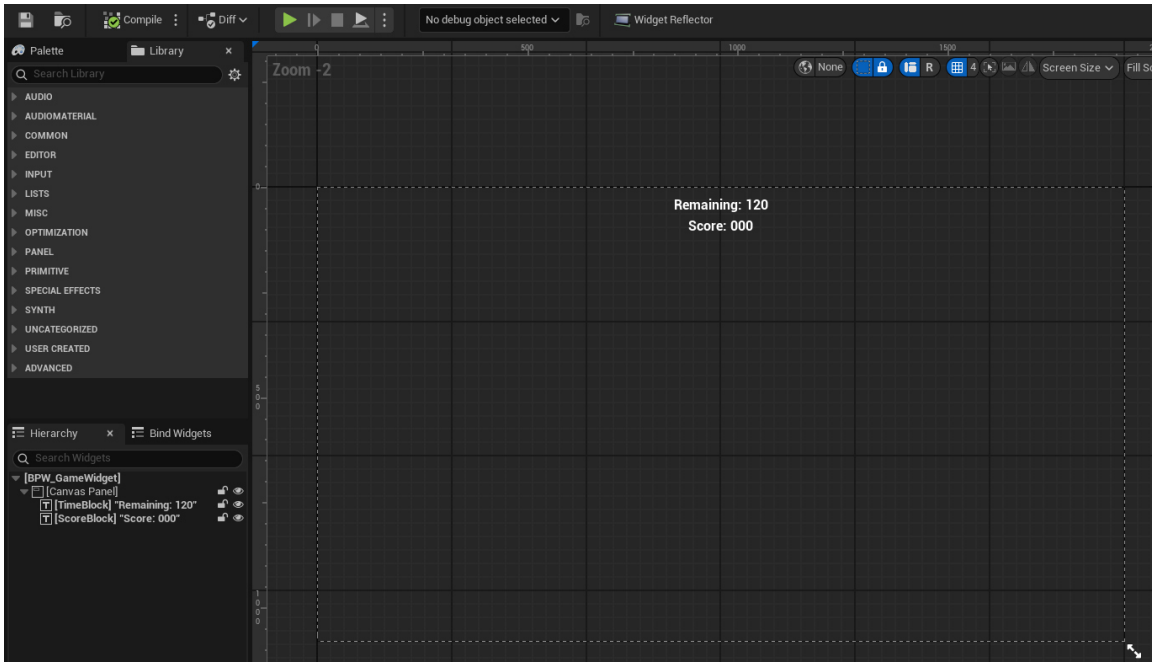

FIGURE 35 HEADS UP DISPLAY WIDGET

The widget for the end screen displays the score and gives the user the options to restart the level or return to the main menu.

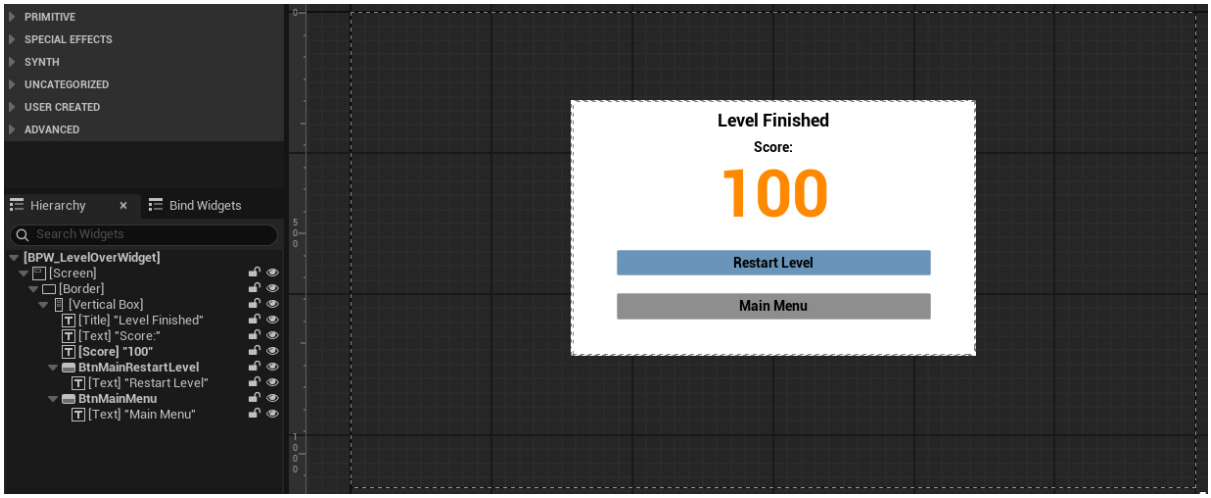

FIGURE 36 LEVEL END SCREEN WIDGET

The widget for the main menu has an option to quit the game as well as options to play the game using 4 maps. The 4 maps to choose from is first, the final terrain that was discussed in the terrain section of this chapter, A level that uses non fractal perlin noise

for the terrain, A level that just uses Worley noise of the terrain and a level that uses fractal perlin noise.

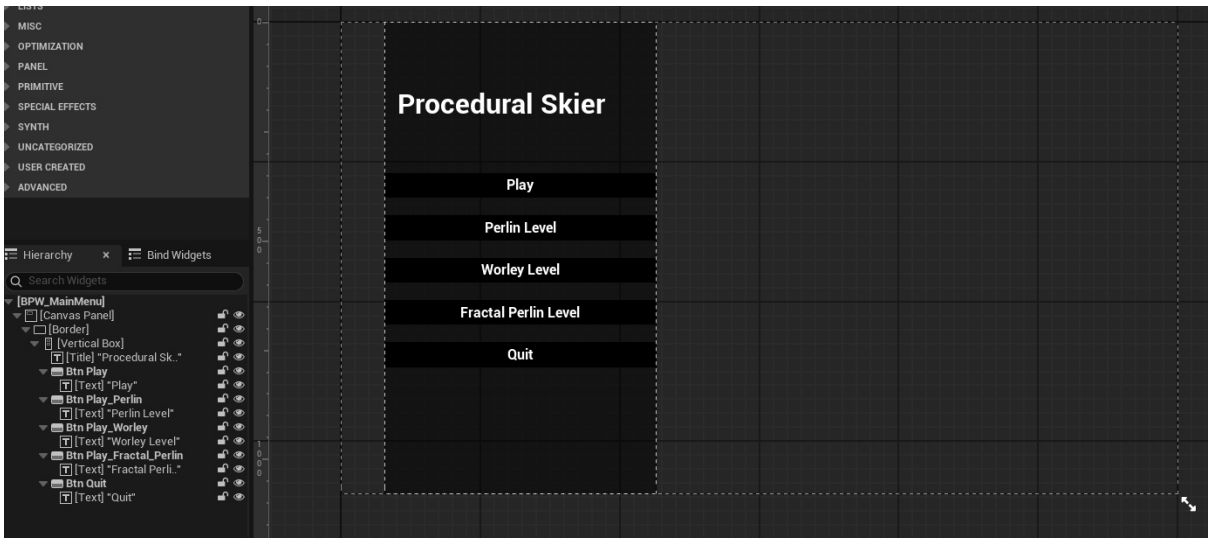

FIGURE 37 MAIN MENU WIDGET

Chapter 5: Evaluation & Conclusion

The aim of this chapter is to provide a summary of what was achieved in this project as well as what lessons were learnt. This section will also highlight what future work this project may lead to.

5.1 Summary of Project

In this project a small video game was created. The main technical focus of the project was to take a game that uses procedural terrain generation to make the levels in the game. In this project terrain was created using 2 different noise algorithms, these algorithms where the Fractal Perlin Noise algorithm and the Worley noise algorithm. The Fractal Perlin Noise algorithm created the main structure as well as the rock terrain and the Worley Noise algorithm created ridges in the terrain. These noise algorithms were separated into 4 different levels, one that uses regular perlin noise terrain, one that uses fractal perlin noise terrain and one that uses Worley noise terrain. Additionally, attempts were made to use a spline-based technique to generate paths in the terrain, but they were not achieved.

A fully functioning physics-based character controller was also created in this project that included a fully working character model which was fully rigged and animated. A gameplay loop was also created with checkpoints and time limits.

An analysis of current work in the field of procedural content generation was also created and summarised via a literature review.

5.2 Critical Evaluation

Although when creating this project the spline-based terrain did not pan out as planned the terrain created though the use of noise algorithms were highly detailed and realistic. Additionally, a lesson was learned about contingency planning and risk management due to this issue. Adaptability was shown in this project as a new gameplay loop was created when the planned gameplay loop became unfeasible when an important feature could not be added.

One issue with this project is that the larger the terrain that was created, the slower the game would run. This could be improved by including level of detail options so that the terrain that is far away from the player is rendered at a lower detail than the terrain that is close to the player. This method has been used in other examples of procedurally generated terrain such as the terrain created by Golubev, Zagarskikh and Karsakov (2016).

Additionally, the height of the player is not set to just above the terrain, so the player has to spawn high up in the sky at the start of the game. Calculating where the ground is before the player spawns and moving them there would solve this issue.

Another note is that controller support could have been added to this project so that it could be assessable to a player who prefer to use a keyboard.

5.3 Future Work

In future this project could be worked on further by finding a way to successfully implement the terrain spline. This could either be done by editing the engine code in Unreal Engine, finding another way to implement this feature or by programming a custom game engine that would be more suited to this task.

Additionally, there are many more noise algorithms that could be used to generate terrain so the topic of noise-based terrain generation could be explored further.

References

42 YEAH., 2023. *Worley and His Noise (Worley Noise/Voronoi Noise)* [online]. Available from: https://blog.42yeah.is/rendering/noise/2023/09/23/voronoi.html [Accessed 4 May 2025].

Blizzard Entertainment, 2016. *Overwatch* [online] Xbox: Blizzard Entertainment.

CHUGANI, V., 2024. *Understanding Euclidean Distance: From Theory to Practice* [Online]. Available from: https://www.datacamp.com/tutorial/euclidean-distance [Accessed 5 May 2025].

DENHAM, T. N.d. *What is UV Mapping & Unwrapping?* [online]. Available from: https://conceptartempire.com/uv-mapping-unwrapping/ [Accessed 4 May 2025].

DOGO'S SCIENCE 2., 2024. *How Does Perlin Noise Work?* [online]. Available from: https://www.youtube.com/watch?v=9B89kwHvTN4 [Accessed 3 March 2025].

DUNN, P. N.d. *Hardware Perlin Noise Demonstration* [online]. Available from: https://dunnbypaul.net/perlin/ [Accessed 3 March 2025].

EPIC GAMES. N.d. *FMath::PerlinNoise2D* [online]. Available from: https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Runtime/Core/Math/FMath/PerlinNoise2D [Accessed 25 February 2025].

ETHERINGTON, T. R., 2022. Perlin noise as a hierarchical neutral landscape model. *Web Ecology*. 22 (1), pp. 1–6. Available from: https://we.copernicus.org/articles/22/1/2022/.

GOLUBEV, K., ZAGARSKIKH, A. and KARSAKOV, A., 2016. Dijkstra-based Terrain Generation Using Advanced Weight Functions. *Procedia Computer Science*. 101, pp. 152–160. Available from: https://www.sciencedirect.com/science/article/pii/S1877050916326862.

GONOG, L. and ZHOU, Y., 2019. A Review: Generative Adversarial Networks. pp. 505–510.

GONZALEZ, P. and LOWE, J., 2015. *Cellular Noise*. Available from: https://thebookofshaders.com/12/ [Accessed 3 May 2025].

HART, J. C. , 2001. Perlin noise pixel shaders New York, NY, USA: Association for Computing Machinery. pp. 87–94. Available from: https://doi.org/10.1145/383507.383531.

Hello Games, 2016. *No Man's Sky* [online] Steam: Hello Games.

HUANG, Y. and YUAN, X., 2023. StyleTerrain: A novel disentangled generative model for controllable high-quality procedural terrain generation. *Computers & Graphics*. 116.

HYTTINEN, T., MÄKINEN, E. and PORANEN, T., 2017. Terrain synthesis using noise by examples New York, NY, USA: Association for Computing Machinery. pp. 17–25. Available from: https://doi.org/10.1145/3131085.3131099.

Ironhide Game Studio., 2013. *Kingdom Rush: Frontiers* [online]. Steam: Ironhide Game Studio.

JAIN, A., SHARMA, A. and RAJAN, K., S., 2024. Learning Based Infinite Terrain Generation with Level of Detailing. *In: 2024 International Conference on 3D Vision (3DV)*. pp. 1048–1058.

KORA, L., 2007. *Implementation of Perlin Noise on GPU An Independent study* [online]. Available from: https://www.sci.utah.edu/~leenak/IndStudy_reportfall/Perlin%20Noise%20on%20GPU.html [Accessed 5 May 2025].

LAGUE, S., 2016. *Procedural Landmass Generation (E01: Introduction)* [online]. Available from: https://www.youtube.com/watch?v=wbpMiKiSKm8&list=PLFt_AvWsXl0eBW2EiBtl_sxmDtSgZBxB3 [Accessed 8 January 2025].

LIU, S., CHAORAN, L., YUE, L., HENG, M., XIAO, H., YIMING, S., LICONG, W., ZE, C., XIANGHAO, G., HENGTONG, L., YU, D. and QINTING, T., 2019. Automatic generation of tower defense levels using PCG New York, NY, USA: Association for Computing Machinery.

Medagon Industries, 2024. *Lonely Mountains: Snow Riders* [online] Steam: Medagon Industries.

Mojang, 2011. *Minecraft* [online] PC: Mojang.

Ninja Kiwi, 2018. *Bloons Tower Defense 6* [online] Steam: Ninja Kiwi

PECH, A., LAM, C. P. and MASEK, M., 2020. Quantifiable Isovist and Graph-Based Measures for Automatic Evaluation of Different Area Types in Virtual Terrain Generation. *IEEE Access*. 8.

PECK, J., 2024. *Fast Noise Lite* [online]. Available from: https://github.com/Auburn/FastNoiseLite/tree/FastNoise-Legacy [Accessed 4 May 2025].

PERLIN, K., 1985. An image synthesizer. *SIGGRAPH Comput.Graph.* 19 (3), pp. 287–296. Available from: https://doi.org/10.1145/325165.325247.

RAOUF, T. N.d. *Perlin Noise: A Procedural Generation Algorithm* [online]. Available from: https://rtouti.github.io/graphics/perlin-noise-algorithm [Accessed 1 March 2025].

RICHSTUBBSANIMATION., 2023. *Easy and Quick Character Rigging in Blender - Blender Basics Tutorial* [online]. Available from: https://www.youtube.com/watch?v=jlwrswJEFBQ [Accessed 8 March 2025].

ROCKAM., 2020. *Fast Noise Generator* [online]. Available from: https://www.fab.com/listings/c1d444fc-54cc-4f11-8a4a-c0c41112a321 [Accessed 4 May 2025].

STÅLBERG, O., MEREDITH, R., KVALE, M. and Plausible Concept., 2018. *Bad North* [online] Steam: Raw Fury.

SU, S., XU, W., TANG, H., QIN, B. and WANG, X., 2024. Edge-protected IDW-based DEM detail enhancement and 3D terrain visualization. *Computers & Graphics*. 122.

SUMMERVILLE, A., SNODGRASS, S., GUZDIAL, M., HOLMGÅRD, C., HOOVER, A. K., ISAKSEN, A., NEALEN, A. and TOGELIUS, J., 2018. Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games*. 10 (3), pp. 257–270.

TUYTEL, R. N.d.a. *Rock Ground 04* [online]. Available from: https://polyhaven.com/a/rocks_ground_04 [Accessed 30 April 2025].

TUYTEL, R. N.d.b. *Snow 02* [online]. Available from: https://polyhaven.com/a/snow_02 [Accessed 30 April 2025].

UNITY. N.d. *Mathf.PerlinNoise* [online]. Available from: https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Mathf.PerlinNoise.html [Accessed 7 January 2025].

WORLEY, S. , 1996. A cellular texture basis function New York, NY, USA: Association for Computing Machinery. pp. 291–294. Available from: https://doi.org/10.1145/237170.237267.

Appendix

A: Renders of Player 3D Model

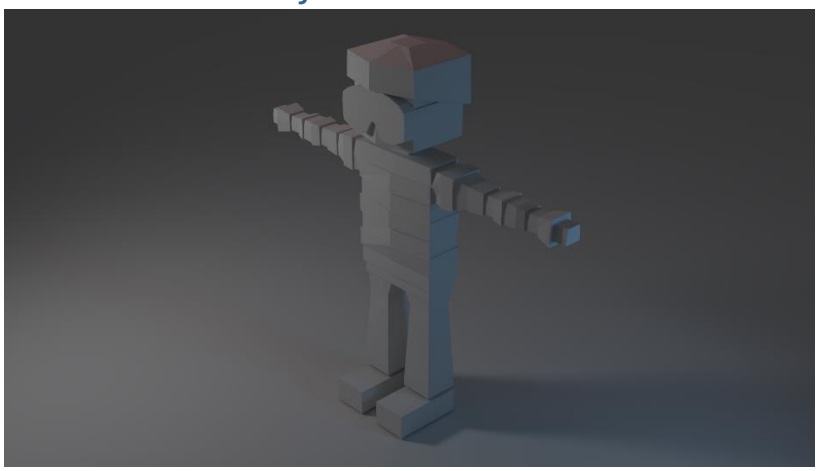

Figure 38 Untextured Skier Model Render

Figure 39 Skier UV Map Render

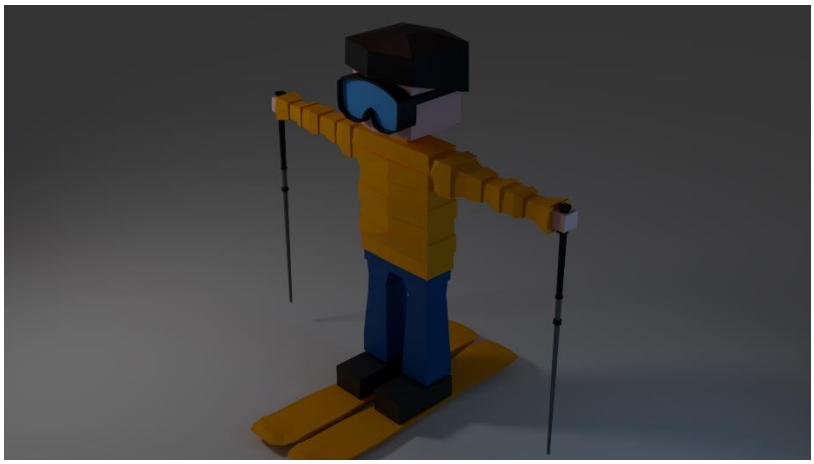

Figure 40 Final Skier Render

B: Instructions for Use

This Project has been created by James Edwards, Student ID 25295039

This project has been created using Unreal Engine 5.5 so it is recommended that it is ran using that version of Unreal Engine.

The first scene that is to be ran is the main menu scene

- - - - - Controls - - - - -

W - Move forward

A - Move Left

D - Move Right

S - Break

- - - - - Objective - - - - -

Collect as many flags as possible within the time limit in each level

- - - - - Levels - - - - -

Main - Level that uses Landscape that is generated using a combination of Fractal Perlin Noise and Worley Noise (Playable by clicking play in the main menu)

Perlin_Level - Level that uses a Landscape generated with a Non-Fractal Perlin Algorithm

Fractal_Perlin_Level - Level that uses a Landscape with the Fractal Perlin Noise Algorithm

Worley_Level - Level that uses Landscape that uses the Worley Noise algorithm